Priority Queues for Common Lisp

Ingvar Mattsson
<ingvar@google.com>

Marco Antoniotti
<marco.antoniotti@unimib.it>

February 4, 2013

Keywords: Heap, Priority Queue, Common Lisp.

1 Introduction

This is a specification for the introduction of a common API for priority queues, also called heaps, in Common Lisp. The specification tries to take into account the common elements present in the several implementations available on the Internet, and to ensure that the API is generic enough to allow for the seamless inclusion of particular flavors of heaps. An inspiration for this specification API is [1], especially w.r.t., the discussion about Heaps and Fibonacci Heaps.

1.1 Rationale

There is no standard heap (or priority queue) implementation in the Common Lisp standard. It is, however, a useful data structure. The intention of this document is to provide a portable, flexible, heap API that can be used on essentially all data where storing according to a ranking criterion makes sense.

This API specification carefully does not discuss how it behaves in a multi-processing environment.

1.2 Guarantees

1.2.1 Time complexity

The heap data structure gives you $O(1)$ peek at one extreme of the heap. It also gives you $O(\lg n)$ addition and removal from the heap.

However, the $O(\lg n)$ insertion and removal relies on an $O(1)$ comparison operator. With having user-specified comparison (and key extraction) operators, the best guarantee the reference implementation can give is that insertion and removal is $O(C \lg n)$ for a comparator complexity of $O(C)$. 
1.2.2 Multi-processing

There are no explicit multi-processing or concurrency guarantees for the generic heaps. However, implementors are encouraged to add recursive locks to each heap object and lock/unlock these as necessary.

1.2.3 Side-effects

Any code that modifies an object currently present in a heap is likely to breach the heap invariant. Doing that is highly discouraged. However, modifying things within an object that does not, in any way, contribute to the value used in comparisons may be safe.

1.3 Design Choices

There are a few design choices to be made when specifying an API for heaps. The following is a list of foreseen issues and their treatment.

1.3.1 Heap Test must be a Total Order

There is no way for a Common Lisp implementation to check and ensure that the function that becomes the heap test (cfr., the constructor make-heap) is a total order (modulo equality). Providing a function that does not represent a total order has undefined consequences.

1.3.2 Equal Keys

The relative order to elements in a heap that admits equal keys is implementation dependent and should not be relied upon.

2 Heaps Dictionary

2.1 Class heap

Class Precedence List:

heap, ..., T

Description:

Any implementation of this specification will provide a class named heap.

Notes:

Each implementation is given the liberty to choose whether to use a structure-class or a standard-class (or another full-blown CLOS class). This implies that specialized heaps can only be derived via single inheritance.
2.2 **Generic Function** heap-p

Syntax:

heap-p object ⇒ generalized-boolean

Arguments and Values:

object – an object.

generalized-boolean – a generalized boolean.

Description:

This function returns NIL when called on a non-heap object and a non-null value if presented with a heap object.

2.3 **Slot Readers** heap-size, heap-total-size, heap-key-function, heap-test-function

Syntax:

heap-size heap ⇒ size
heap-total-size heap ⇒ total-size
heap-key-function heap ⇒ keyfun
heap-test-function heap ⇒ cmpfun

Arguments and Values:

heap – a heap.

heap-key-function – a function designator.

heap-test-function – a function designator.

size – a (positive) integer.

total-size – a (positive) integer.

Description:

The heap-size and heap-total-size return the number of elements in the heap.

The heap-key-function and heap-test-function accessors return the test function and the key function used by the heap implementation to maintain the heap invariant.
2.4 Type heap-finger

Many operations on heaps require to “change” something that is located in a certain “position” in the underlying data structure. To support these operations the specification requires implementations to provide an opaque type named heap-finger, i.e., to provide a way to keep a “finger” on a certain position within the heap\(^1\).

As an example, a traditional implementation of heaps based on arrays could define heap-finger as

```
(deftype heap-finger () 'fixnum)
```

Notes:

This specification does not prescribe anything in particular regarding the behavior of heap-fingers and the garbage collector. An implementation is free to add a :weak key to the make-heap constructor (see below) and to return a weak heap-finger, that works well with the garbage collector.

2.5 Function heap-finger-p

Syntax:

```
heap-finger-p object ⇒ boolean
```

Arguments and Values:

- \(object\) – an object.
- \(boolean\) – a boolean.

Description:

Returns T if \(object\) is a heap-finger, NIL otherwise.

2.6 Condition heap-error

Class Precedence List:

```
heap-error, simple-error, ..., T
```

Description:

The root of specialized errors raised by the heap operations; the heap for which the error is being signaled can be initialized with the keyword :heap and can be read by the accessor heap-error-heap. The default for the underlying slot is NIL.

\(^1\)The term “finger” has been extensively used in the algorithms and data structure literature.
See Also:
heap-error-heap.

2.7 Function heap-error-heap

Syntax:
heap-error-heap heap-error ⇒ heap

Arguments and Values:
heap-error – a heap-error
heap – a heap.

Description:
Returns the heap associated to the condition heap-error or NIL if the slot is uninitialized.

2.8 Condition empty-heap-error

Class Precedence List:
empty-heap-error, heap-error, ..., T

Description:
The condition that may be signaled when certain operations are attempted on an empty heap.

See Also:
heap-error-heap, heap-error.

2.9 Condition invalid-heap-finger-error

Class Precedence List:
invalid-heap-finger-error, heap-error, cell-error, ..., T

Description:
The condition that may be signaled when certain operations are attempted on an invalid “position” in a heap. The offending finger must be passed at initialization time with the keyword :name.
See Also:
heap-error-heap, heap-error, heap-finger.

Notes:
invalid-heap-finger-error inherits from cell-error, hence, cell-error-name is used to get the offending finger.

2.10 Condition invalid-key-error

Class Precedence List:
invalid-key-error, heap-error, ..., T

Description:
The condition that may be signaled when certain operations are attempted with an invalid “key” in a heap. The offending key is initialized using the :offender keyword and can be retrieved by the invalid-key-error-offender function.

See Also:
invalid-key-error-offender, heap-error-heap, heap-error.

2.11 Function invalid-key-error-offender

Syntax:
invalid-key-error-offender i-k-e ⇒ key-object

Arguments and Values:
i-k-e – a invalid-key-error.

key-object – a object.

Description:
Given an instance of invalid-key-error, invalid-key-error-offender returns the offending key-object associated with i-k-e.

2.12 Function make-heap

Syntax:
make-heap &key test key initial-size class initial-contents &allow-other-keys ⇒ heap
Arguments and Values:

test — a function designator for a binary function returning a generalized boolean; default is <.

key — an accessor for an object; default is identity.

initial-size — a positive fixnum; default is 16.

class — a class designator; the default is heap.

heap — an instance of the heap class or of any of its descendant classes.

Description:
Returns a newly created heap, using the specified test as the heap criterion, using key to extract the values to be compared.

2.13 Generic Function empty-heap-p

Syntax:
empty-heap-p heap ⇒ boolean

Arguments and Values:
heap — a heap.
boolean — a boolean.

Description:
This function returns T when called on an empty heap, NIL otherwise.

2.14 Generic Function full-heap-p

Syntax:
full-heap-p heap ⇒ boolean

Arguments and Values:
heap — a heap.
boolean — a boolean.
Description:
This function returns T when no more values can be inserted in the heap, NIL otherwise.
Certain versions of heaps are only limited by the system’s memory limitations. In these cases full-heap-p always returns NIL. Implementations are required to document these cases.

2.15 Generic Function insert

Syntax:
insert heap value ⇒ value finger

Arguments and Values:
heap – a heap.
value – an object.
finger – a heap-finger.

Description:
Inserts a new value into the heap. The value inserted is returned alongside the “location”, pointed by finger in which it was inserted.

2.16 Generic Functions extract, extract-from

Syntax:
extract heap &optional default error-if-empty ⇒ value
extract-from heap finger &optional default ⇒ value

Arguments and Values:
heap – a heap.
finger – a heap-finger.
default – an object; default is NIL.
error-if-empty – a generalized boolean; default is NIL.
value – an object.
Description:

`extract` removes and returns the value at the top of the heap, unless the heap is empty. If the heap is empty and `error-if-empty` is `NIL`, `default` is returned; otherwise an `empty-heap-error` error is signaled.

`extract-from` removes and returns the value present in the heap in “position” `finger`. If the `finger` is invalid and `error-if-empty` is `NIL`, `default` is returned; otherwise an `invalid-heap-finger-error` error is signaled.

Exceptional Situations:

The errors `empty-heap-error` and `invalid-heap-finger-error` are signaled in the case described above.

2.17 Generic Function `peek`

Syntax:

```
peek heap &optional default error-if-empty ⇒ value
```

Arguments and Values:

- `heap` – a heap.
- `default` – an object; default is `NIL`.
- `error-if-empty` – a generalized boolean; default is `NIL`.
- `value` – an object.

Description:

Returns the value at the top of the heap, without modifying the heap. If the heap is empty and `error-if-empty` is `NIL`, `default` is returned; otherwise an error of type `empty-heap-error` is signaled.

See Also:

`empty-heap-error`

2.18 Generic Functions `change-key`, `decrease-key`, `increase-key`

Syntax:

```
change-key heap new-key finger ⇒ heap old-key new-finger
decrease-key heap new-key finger ⇒ heap old-key new-finger
increase-key heap new-key finger ⇒ heap old-key new-finger
```
Arguments and Values:

heap – a heap.
new-key – an object.
finger – a heap-finger.
old-key – an object.
new-finger – a heap-finger.

Description:

change-key changes the key corresponding to the heap entry at position finger with new-key; the heap is restructured as a consequence. The three values returned are the restructured heap, the key (old-key) used before the change-key had any effect on the heap structure, and the new-finger resulting after the changes effected by change-key.

The generic functions decrease-key and increase-key, check that new-key is, respectively, “smaller” or “greater” than old-key (the key associated to finger). If the check succeeds, then the effect of the call is that of calling change-key. If the check fails than an error of type invalid-key-error is signaled.

See Also:

invalid-key-error.

Notes:

It is assumed that all implementations will actually wrap the actual heap internal data structure in a container shell of some kind. I.e., the heap is returned as such, with only the inside structures changed as a consequence of change-key.

2.19 Generic Function fix-heap

Syntax:

fix-heap heap finger ⇒ heap new-finger

Arguments and Values:

heap – a heap.
finger – a heap-finger.
new-finger – a heap-finger.
Description:

This function is used to fix the heap invariant starting from a given finger. This function should be used after changes to an object stored in the heap affecting the heap invariant (cfr., (setf value-at)).

See Also:

(setf value-at).

2.20  Generic Functions key-at, value-at,
       content-at, content-at*

Syntax:

key-at heap finger ⇒ key
value-at heap finger ⇒ value
(setf value-at) value heap finger ⇒ value
content-at heap finger ⇒ key, value
content-at* heap finger ⇒ content

Arguments and Values:

heap — a heap.
finger — a heap-finger.
key — an object.
old-key — an object.
value — an object.
content — a cons of the form (key . value).

Description:

As the names imply, key-at returns the key that can be found in the heap in correspondence of the finger. value-at returns the value that can be found in the heap in correspondence of the finger. The setf form can be used to modify what is associated to key in correspondence of the finger. No change in the underlying heap structure is required. Therefore, in order to ensure that the heap invariants are maintained after a (setf value-at) the user may have to call fix explicitly.

content-at returns two values: the key and the value that can be found in the heap in correspondence of the finger. content-at* behaves like content-at but it returns a dotted pair (key . value).
See Also:

fix-heap.

Notes:

Problems with (setf content-at) may arise when heap-key-function is identity or conceivably similar cases. When this happens, then (setf content-at) may violate the heap invariant.

2.21 Generic Functions merge-heaps, nmerge-heaps

Syntax:

merge-heaps heap1 heap2 &key &allow-other-keys ⇒ new-heap
nmerge-heaps heap1 heap2 &key &allow-other-keys ⇒ new-heap

Arguments and Values:

heap1 – a heap.
heap2 – a heap.
new-heap – a heap.

Description:

merge-heaps constructs a new-heap that contains all the values of heap1 and heap2. The nmerge-heaps may destructively modify either heap1 or heap2 (or both) and may return either in lieu of new-heap.

Notes:

It is understood that the performance guarantees for this operation depend on the underlying implementation.

2.22 Generic Functions heap-keys, heap-values, heap-contents

Syntax:

heap-keys heap &optional (result-type ’list) ⇒ result
heap-values heap &optional (result-type ’list) ⇒ result
heap-contents heap &optional (result-type ’list) ⇒ result
Arguments and Values:

*heap* – a heap.

*result-type* – a designator for a *sequence* type.

*result* – a sequence of type *result-type*

Description:

*heap-keys* returns a sequence of *result-type* containing the *keys* in the *heap*.

*heap-values* returns a sequence of *result-type* containing the *values* in the *heap*.

*heap-contents* returns a sequence of *result-type* containing pairs (*key* . *value*) in the *heap*; i.e., with the default *result-type* of *list*, *result* is a *association list*.

Exceptional Situations:

A *type-error* is signaled if *result* cannot be coerced to a sequence of type *result-type*.

Notes:

The content of *result* is not affected by interleaving *change-key’s*. Users cannot make assumptions on the behavior.

References


A Copying and License

This work may be distributed and/or modified under the conditions of the *LaTeX Project Public License* (LPPL), either version 1.3 of this license or (at your option) any later version. The latest version of this license is in http://www.latex-project.org/lppl.txt and version 1.3 or later is part of all distributions of LaTeX version 2005/12/01 or later.

This work has the LPPL maintenance status ‘maintained’. The Current Maintainer of this work is Marco Antoniotti <marco.antoniotti@unimib.it>.